A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications

Jordi Caubet[1], Judit Gimenez[1], Jesus Labarta[*1], Luiz DeRose[2], and Jeffrey Vetter[**3]

[1] European Center for Parallelism of Barcelona
Department of Computer Architecture
Technical University of Catalonia
Barcelona, Spain
{jordics, judit, jesus}@cepba.upc.es
[2] Advanced Computing Technology Center
IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
laderose@us.ibm.com
[3] Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA
vetter@llnl.gov

Abstract. In this paper we present OMPtrace, a dynamic tracing mechanism that combines traditional tracing with dynamic instrumentation and access to hardware performance counters to create a powerful tool for performance analysis and optimization of OpenMP applications. Performance data collected with OMPtrace is used as input to the Paraver visualization tool for detailed analysis of the parallel behavior of the application. We demonstrate the usefulness of OMPtrace and the power of Paraver for tuning OpenMP applications with a case study running the US DOE ASCI Sweep3D benchmark on the IBM SP system at the Lawrence Livermore National Laboratory.

1 Introduction

OpenMP has emerged as the standard for shared memory parallel programming, allowing users to write applications that are portable across most shared memory multiprocessors. However, in order to achieve high performance on these systems, application developers still face a large number of application performance problems, such as load imbalance and false sharing. These performance

[*] This work was partially supported by IBM under a Shared University Research grant and by the Spanish Ministry of Education (CICYT) under contract TIC98-0511
[**] This work was performed under the auspices of the U.S. Dept. of Energy by University of California LLNL under contract W-7405-Eng-48. LLNL Document Number UCRL-JC-142770.

problems make application tuning complex and often counter-intuitive. Moreover, these problems are hard to detect without the help of performance tools that have low intrusion cost and are able to correlate dynamic performance data from both software and hardware measurements.

In this paper, we describe OMPtrace - a dynamic tracing mechanism that combines traditional tracing with dynamic instrumentation and access to hardware performance counters - to create a powerful tool for performance analysis and optimization of OpenMP applications.

OMPtrace is built on top of the Dynamic Probe Class Library (DPCL)[2], an object-based C++ class library and runtime infrastructure that flexibly supports the generation of arbitrary instrumentation, without requiring access to the source code. DPCL allows the instrumentation of the OpenMP runtime systems, providing the flexibility to measure the overhead of initialization and finalization of parallel regions. For detailed analysis of the parallel behavior of the application, OMPtrace data is, then, analyzed with the Paraver visualization tool[3]. To demonstrate the power of OMPtrace and Paraver, we analyze the performance of the Sweep3D application[4] as a case study.

The remainder of this paper is organized as follows. In Section 2 we briefly describe the main DPCL issues. In Section 3 we discuss the OMPtrace interface. In Section 4 we present a case study where we describe the steps followed to analyze the Sweep3D application and how OMPtrace and Paraver were useful to identify potential improvements. Finally, our conclusions are summarized in Section 5.

2 The Dynamic Probe Class Library

Traditionally, instrumentation systems have had to strike a balance between minimizing instrumentation overhead and maximizing the amount of performance data captured. One approach to managing instrumentation overhead is to limit both the number of events recorded and the size of those events. However, this could mean that key events may not have been recorded. Likewise, if too much instrumentation is inserted, the overhead may be so high that it is no longer representative of the un-instrumented program's execution behavior.

Another challenge is that many instrumentation systems require that programs be re-compiled after being instrumented. While this is generally possible, for large applications it can be time consuming. Even worse, for third party libraries and applications users where the source code may not be available, re-compiling will not be possible. An alternative is to allow a program to be modified while it is executing, and thereby eliminate the need to re-compile, re-link, or re-execute the program.

Dynamic instrumentation provides the flexibility for tools to insert probes into applications only where it is needed. The Dynamic Probe Class Library, developed at IBM, is an extension of the dynamic instrumentation approach, pioneered by the Paradyn group at the University of Wisconsin[5]. DPCL is built on top of the Dyninst Application Program Interface (API)[1]. Using DPCL,

a performance tool can attach to an application, insert code patches into the binary and start or continue its execution. Access to the source code of the target application is not required and the program being modified does not need to be re-compiled, re-linked, or even re-started.

DPCL provides a set of C++ classes that allows tools to connect, examine, and instrument a spectrum of applications: single processes to large parallel applications. DPCL is composed of a client library, a runtime library, a daemon, and a super-daemon. End user tools can be created with the client library. The runtime library supports instrumentation generation and communication. The daemon interfaces with the Dyninst library to instrument and manage user processes; a super-daemon manages security and client connections to these DPCL daemons. With DPCL, program instrumentation can be done at function entry points, exit points, and call sites.

3 The OMPtrace

The integration of DPCL into OMPtrace is based on the fact that the IBM compiler translates OpenMP directives into function calls. Figure 1 shows, as an example, the compiler transformations for an OpenMP parallel loop. The OpenMP directive is translated into a call to a function from the OpenMP runtime library ("`xlsmpParDoSetup`"), which is responsible for thread management, work distribution, and thread synchronization. The loop is transformed into a function ("`A@OL1`" in the example in Figure1) that is called by each of the OpenMP threads.

Since DPCL allows the installation of probes at function call entry and exit points, as well as before and after a function call, the OMPtrace tool installs two pairs of probes for each parallel region in the target application. As shown in Figure 2 the first pair (DPCL probe (1)) is inserted before and after the call to the OpenMP runtime library function, while the second pair (DPCL probe(2)) is inserted at the call entry and exit point of the parallel region. Given these two pairs of probes, one can measure the overhead of starting and terminating a parallel region. Additionally, a third pair of probes (DPCL probe (3)) is inserted at the call entry and exit points of each function that contains an OpenMP parallel region.

Figure 3 displays the startup procedure executed by OMPtrace. The tool communicates with the DPCL daemon (1), which in turn acts on the application binary (2). Probe installation (3) is executed in two steps. First, OMPtrace requests the DPCL communication daemon to load the tracing module into the target application. This module contains the functions that will be called by the probes. Once the tracing module has been loaded, OMPtrace requests the communication daemon to inserts the probes into the application. After the probes are installed, OMPtrace starts the application. Notice that nothing precludes OMPtrace from attaching to a running application and execute the same procedure. We are considering this feature for future work.

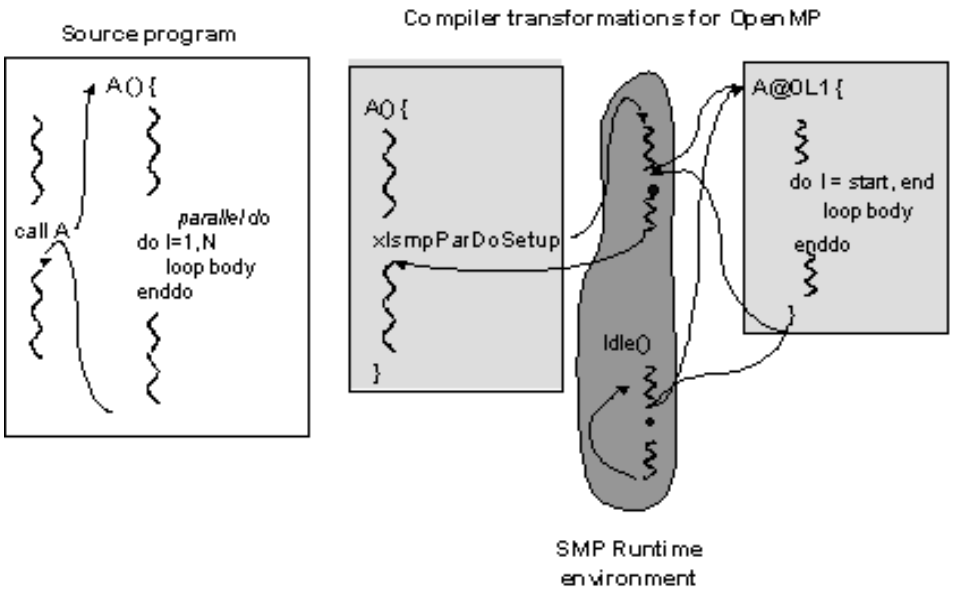

Fig. 1. Compiler transformations for an OpenMP parallel loop.

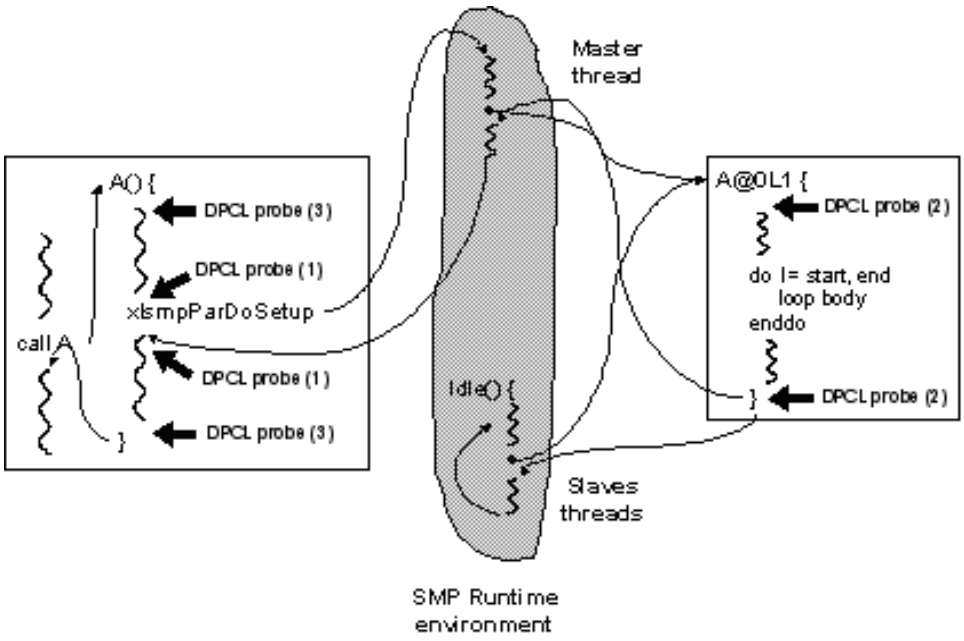

Fig. 2. DPCL probes on functions that contain parallel regions.

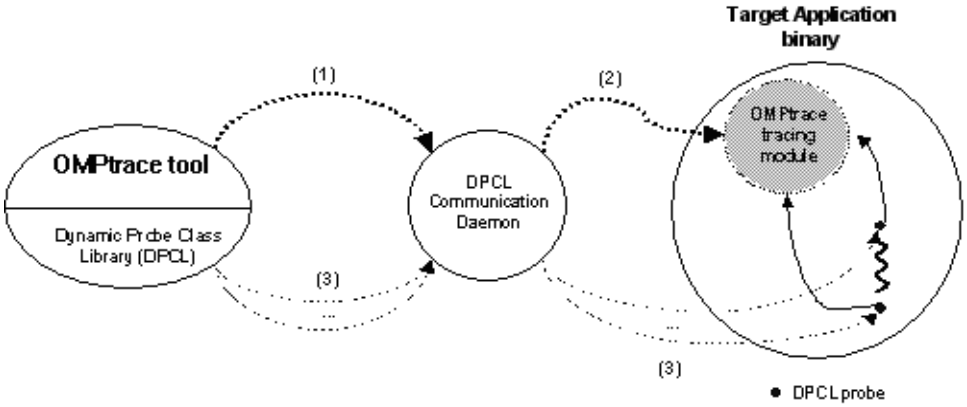

Fig. 3. Application startup with OMPtrace.

Similarly, OMPtrace can be used to instrument locks that are used to ensure mutual exclusion in the application. In this case, dynamic instrumentation is placed before and after the OpenMP functions that are called to handle the locks, generating an event trace every time that a thread enters in one of the following four states: trying to acquire a lock, lock acquisition, starting to release a lock, and lock release. This instrumentation is useful to measure lock overhead, contention in critical sections, and the actual pattern of lock acquisition. Since this instrumentation may introduce a significant overhead, especially for very small critical sections, it is only activated when specified by the user with command line flags when executing OMPtrace. It is our experience that even in cases where the overhead is significant, the information on the pattern and interaction between threads that this tracing facility provides is very helpful to improve the performance of the parallel program.

Another OMPtrace feature is the ability to automatically access hardware counters. The IBM Power3 processor provides 8 counters, each one able to count a number of hardware events. OMPtrace allows users to select any valid combination of hardware events, via an environment variable. By default, OMPtrace uses a standard set of events to count instructions, floating point operations, fused multiply adds (FMAs), and loads. When the hardware counters option is activated, OMPtrace emits event records at the entry and exit of every instrumented point in the program, identifying the hardware events being collected and for each event, the count between the current and the previous tracing point. In order to avoid excessive overhead and reduce trace file size for the default analysis, this feature is also only activated via a command line flag.

One of the known weakness of hardware performance counters is that they only provide raw counts, which does not necessarily help users to identify which events are responsible for bottlenecks in the program performance. However, Paraver has a very flexible mechanism to compute and display a large number

of performance indices and derived metrics from the information emitted into the trace by OMPtrace. Thus, the hardware counter information included in the trace file can be later processed by Paraver to generate a large number of performance indices, which allows users to correlate the behavior of the application to one or more of the components of the hardware. For example, Paraver can display as a function of time for a given routine (or interval) the quotient between the number of L1 misses (as reported by the event at the exit of the routine) and the duration of the routine. Indices such as L1 misses per second or floating point operations per second can be visualized as a function of time. Additionally, a second level of semantic functions can be obtained by combining (i.e., adding, dividing, etc.) the functions of time computed directly from the records in the trace as stated above. We call this feature *derived windows*. For example, starting with a window that looks at "*cycles*" to compute the number of cycles for each function (or interval) and other window that looks at the "*instructions*" it is possible to derive an IPC window by dividing those two windows. This derived window will display the actual Instructions per Cycle obtained for each interval of the application, which can be useful for example to compare with the theoretical limit of the machine (4 issue in the Power3 case).

An interesting way to use these derived windows is to build performance models of the processor and try to explain the performance of the application based on these models. For example, one can compute the theoretical IPC limit considering just the number of floating point operations and the number of misses by taking into account that only two FPUs are available and assuming a certain miss cost. Comparing this model with the observed IPC gives an insight on whether the performance is limited by the number of FPUs or by the cost of the cache miss.

In addition to installing these dynamic probes, OMPtrace accepts static instrumentation placed by the user, for tracing of other functions or code regions in the program. During program execution, OMPtrace generates trace records. These records contain absolute times from the activation of the instrumented points in the program during the parallel execution, as well as, the information gathered for these points (for example, data from hardware performance counters). Each record represents an event or activity associated to one thread in the system. At the end of execution, these traces are combined into a single Paraver trace file, in order to convert these "*punctual*" events into "*interval values*".

4 Case Study

In this section we describe the steps followed to analyze an application and how OMPtrace and Paraver were useful to identify potential improvements. We observe that performance tuning of any large application is in general a never-ending task, with new potential improvements arising just after a previous one has been implemented. Thus, our intent was not to optimize the performance of the application to the utmost possible level. Instead, we focused on the way Paraver was helpful in the process.

In this case study, we used the US DOE ASCI Sweep3D benchmark, which uses a multidimensional wavefront algorithm for "discrete ordinates" deterministic particle transport simulation. Figure 4 displays the Sweep3D major data structures and the iteration space. The core computation presents reductions in all directions (i, j, k, and m); thus posing some problems to parallelization. To solve these problems, Sweep3D benefits from multiple wavefronts in multiple dimensions, which are partitioned and pipelined on a distributed memory system. The three dimensional space is decomposed onto a two-dimensional orthogonal mesh, where each processor is assigned one columnar domain, as shown on Figure 5(a). Sweep3D pipelines the `K` dimension, exchanging messages between processors as wavefronts propagate diagonally across this 3D space in eight directions.

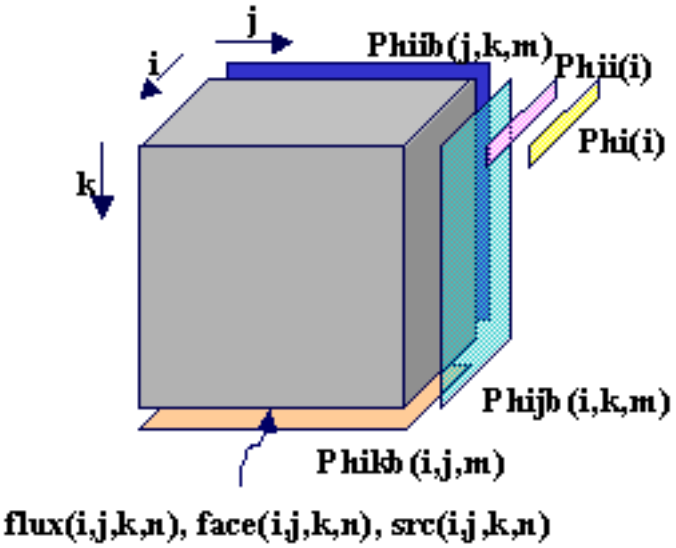

Fig. 4. Sweep3D major data structures and iteration space.

Within each MPI process domain further decomposition of work among several threads can be achieved with OpenMP. The approach is to parallelize the execution of the planes of a diagonal wavefront that traverses the sub-cube computed by each MPI process. Each such plane is inherently parallel as each of its points contributes to a different reduction in each of the i, j, and k directions, as shown in Figure 5(b). This is nevertheless at the expense of additional index computations and triangular loop trip count, which causes significant overhead both in terms of index computations and of OpenMP run time library overhead.

Figure 6 displays the computational flow of Sweep 3D in its original version, which we will refer here as "*diag*" version. However, as an alternative approach

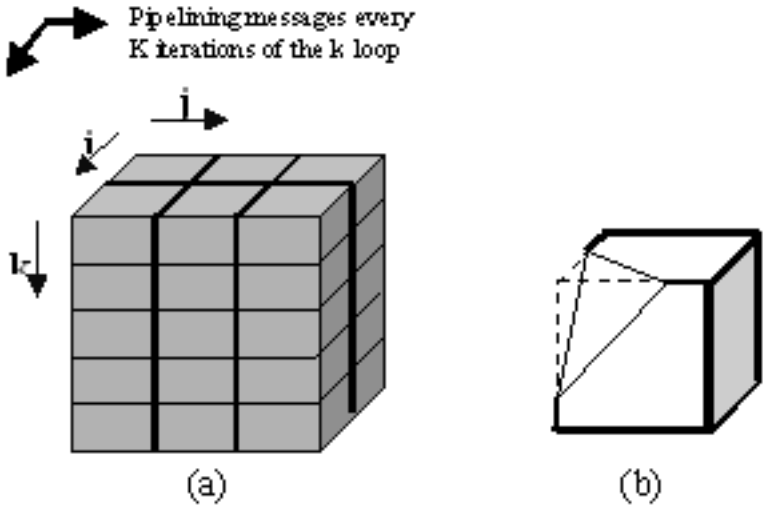

Fig. 5. (a) MPI parallelization structure and (b) sub-cube diagonal parallelization structure with OpenMP.

described in the source distribution, the "`do idiag`" and "`do jkm`" loops, shown in Figure 6, can be replaced by a tripled nested loop ("`do m`", "`do k`", and "`do j`"), which we will refer here as "*mkj*" version.

Our trace collection and analysis was performed with a small problem size, using a cube of dimensions 50×50×50, running on one SP Nighthawk II node with 16 375 MHz Power3+ processors. The performance observations were then validated running a mixed MPI/OpenMP code with a larger problem size, using a cube of dimensions 300×300×100, on 12 SP Nighthawk I Nodes, each node with 8 222 MHz Power3 processors.

As described above, the original MPI version is parallelized in two levels, along the "I" and "J" dimensions. Table 1 presents the elapsed times in seconds for the MPI versions corresponding to different partitioning of the global iteration space, and the elapsed time for the OpenMP "diag" version. The first number in the decomposition indicates the number of processors used for the partitioning across the I dimension, while the second number indicates the number of processors for the J dimension.

We observed that on 6 processors, the best MPI decomposition ran in 3.69 seconds, while the OpenMP version ran in 7.78 seconds. The OpenMP performance with 12 processors was almost three times worst than the best MPI performance. In order to identify the reasons for this performance difference, we obtained two sets of trace of the MPI and the OpenMP versions, one using the default set of hardware counters to measure communication and synchronization overheads, and the other counting cache misses (level 1 and 2) and TLB misses to investigate locality issues.

```
DO iq=1,8                          ! octants
  DO mo=1,mmo                      ! angle pipelining loop
    DO kk=1,kb                     ! k-plane pipelining loop
      RECV E/W                     ! recv block I-inflows
      RECV N/S                     ! recv block J-inflows
      DO idiag=1,jt+nk-1+mmi-1     ! JK-diagonals with MMI pipelining
        DO jkm=1,ndiag             ! I-lines on this diagonal
          j,k,m = f(idiag,jkm)     ! map to j, k, and m indices
          DO i=1,it                ! source (from Pn moments)
          ENDDO
          DO i=i0,i1,i2            ! Sn eqn
          ENDDO
          DO i=1,it                ! flux (Pn moments)
          ENDDO
          DO i=1,it                ! DSA face currents
          ENDDO
        ENDDO
      ENDDO
      SEND E/W                     ! send block I-outflows
      SEND N/S                     ! send block J-outflows
    ENDDO
  ENDDO
ENDDO
```

Fig. 6. Sweep 3D control flow

NB Domains	OpenMP time	Decomposition	MPI time
6	7.78	1x6	3.97
		2x3	3.69
		3x2	3.71
		6x1	4.47
12	6.55	1x12	3.50
		2x6	2.74
		3x4	2.21
		4x3	2.25
		6x2	2.97
		12x1	3.98

Table 1. Elapsed time in seconds for "diag", running the small problem with OpenMP and with MPI.

Using Paraver to compute the total useful computation, we observed that both versions were loosing a similar percentage of time in synchronization and communication. The percentage of time inside numerical computation routines was around 65% for both runs. In the case of OpenMP this low percentage was partially due to the fine granularity of the triangular loops, and because it still executed some sequential computation, since only the major computational loop was parallelized. On the other hand, we observed that the OpenMP version had less L1 misses (2137 per ms) and TLB misses than the MPI version (4153 L1 misses per ms), but much more L2 misses (1356 per ms for the OpenMP version versus only 133 per ms for the MPI version). The rate of L2 misses per millisecond for one traversal of the 3D iteration space in "diag" is shown in Figure 7 and Figure 8 for the OpenMP and MPI versions respectively. In these figures, darker gray (blue) represents large values, while white (green) corresponds to low values. The areas with low values in Figure 7 correspond to intervals where the threads are waiting for work or synchronizing. Hence, our optimization efforts concentrated on improving locality of the OpenMP version and minimizing coherence invalidations.

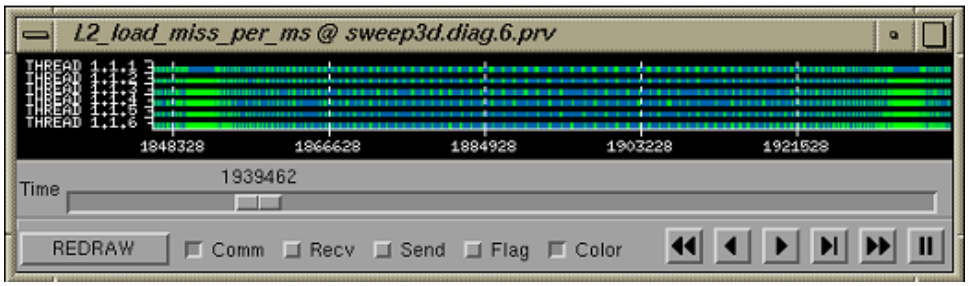

Fig. 7. L2 misses per milliseconds for the OpenMP execution of "diag".

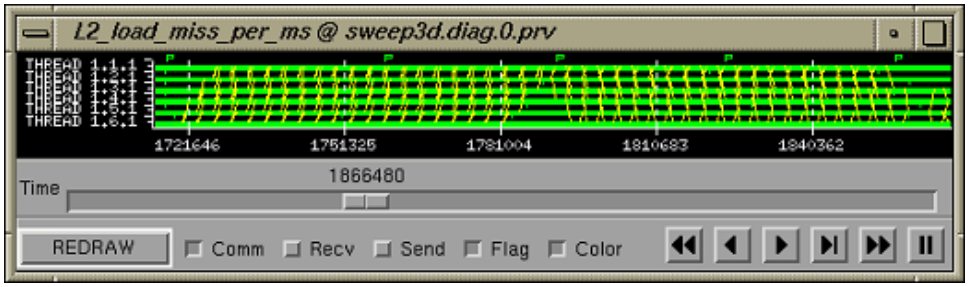

Fig. 8. L2 misses per milliseconds for the MPI execution of "diag".

Taking into account that shared memory is inherently more efficient than message passing for fine grain synchronization, we implemented an OpenMP parallel version based on the "mkj" version, where the outer loop was parallelized and the internal precedence was enforced by some synchronization mechanism. Two approaches were implemented. The first was the version "*ccrit*", which uses the `CRITICAL` OpenMP directive for the implementation of the reduction. The result was a fairly high contention on the lock, a behavior that could be visualized with Paraver, as shown in Figure 9, which displays the behavior of the critical section access. The long regions in gray (red) correspond to the time threads are trying to get a lock that is already taken. The dark (blue) periods correspond to threads using the lock. White (green) is when a thread releases the lock and light gray (light blue) corresponds to execution outside of the critical section. As can be observed, the sequence of accesses does not follow a specific pattern, and the waiting time to obtain a lock has a large variance. Using the quantitative analysis module of Paraver we measured the average waiting time to be 170 microseconds with a standard deviation of 172.

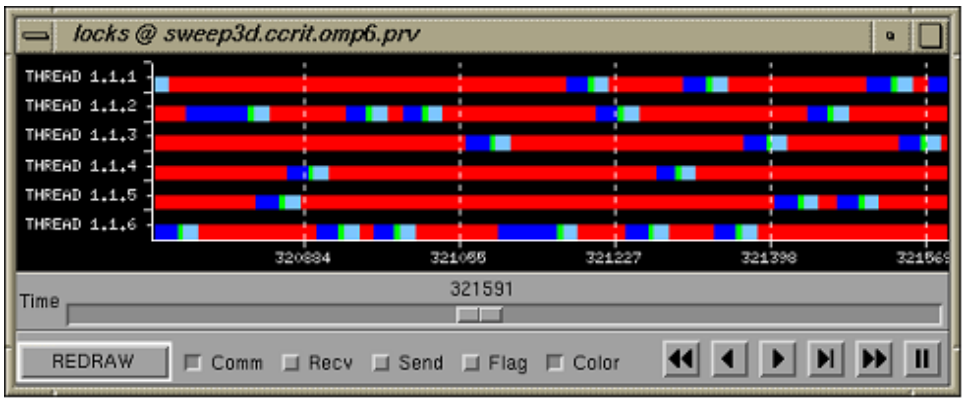

Fig. 9. Lock access pattern in the "ccrit" version.

The second synchronization mechanism, which we refer to "*cpipe*", was implemented based on shared arrays and busy waiting. In this version the iterations of the parallelized loop ("`do m`") are interleaved across threads. When a thread finalizes its part of computation involved in the reduction it signals to the following thread to continue with its part. In this approach, the reductions are executed in sequential order (although different threads compute different parts). Thus, in this case, since the sequential order of the computation of the reductions are preserved, the numerical results are identical to the sequential execution, independently of the number of processors used in the parallel computation. This is an important advantage compared to the "ccrit" version, where the critical sections used to assure atomicity of the reduction updates do not preserved their order; thus resulting in numerical differences between runs. In this

version we observed that the synchronization overhead was very low and a fairly good pipelining was achieved.

A comparison of the single processor run of the "diag" and "mkj" versions showed that the "mkj" version was significantly slower. Using Paraver traces with the sequential application we observed that the first, third and fourth "`do i`" loops were touching the variables "`flux`", "`src`", and "`face`" and incurred most of the level 2 cache misses. However, analyzing the source code, it could be observed that interchanging the "`do m`" loop inwards would reduce misses in the "`do i`" loops. Besides the locality problems, parallelizing the m dimension also has the problem of the small trip count of the loop (only 6), which limits parallelism. Taking into account data locality and trip count considerations described above, we interchanged the loops, creating the version "*kjmi*". In order to achieve good pipelining overlap in this version, the "`do k`" loop was parallelized with `SCHEDULE(STATIC,1)`, which means totally interleaved. This causes matrix "`Phikb`" to be circulated between processors for each k, generating level 2 cache invalidations and a slightly higher miss ratio than the MPI version. Thus, in order to increase the reuse of "`Phikb`" we introduced a final modification (version "*Kjkmi*") where the k loop was strip-mined and interchanged according to the version name.

version	1	2	3	4	5	6	8	9	10	11	12	13	14	15	16
ccrit	28.26	24.41	26.84	26.47	29.28	30.34					30.43				
cpipe	25.63	18.45	13.01	12.53	10.06	7.67					7.76				
diag	17.28	13.09	11.40	9.64	8.50	7.78					6.55				
kjmi	14.86	10.01	7.35	5.82	4.89	4.34	3.62	3.38	3.09	3.04	2.88	2.69	2.69	2.64	2.53
Kjkmi	14.91	8.47	6.35	4.91	4.24	3.58	2.90	2.81	2.78	2.65	2.29	2.22	2.16	2.19	2.15

Table 2. Elapsed time in seconds for the different OpenMP versions.

A performance summary of the different OpenMP versions of the program, running the small problem size is presented in Table 2. The numbers show the inefficiency and lack of scalability of the "ccrit" version. The overhead of the mutex lock and unlock needed to protect the critical section can be observed, when comparing the times for just one thread in the "ccrit" and "cpipe" versions. The huge contention at the lock shown in Figure 9 causes the scalability problems.

Although "cpipe" performed better than "ccrit", when comparing to the other three versions we observe that "cpipe" also had poor locality behavior, and scalability problems. These problems occur mainly because the parallel loop on "m" has only an iteration count of 6, which results in a poor pipelining.

The scalability of "diag" is limited, as mentioned above, due to the very high number of L2 misses caused by false sharing and the variable trip count of the parallelized loops. The overhead of opening and closing such parallel loops with

very small trip counts for the diagonal planes at the corners of the cube also contributes to the poor scalability of this version.

The two final versions show much better behavior for just one thread, an effect that not only benefits the OpenMP code, but also the MPI. Scalability is fair, and the performance achieved is equivalent to that of pure MPI as reported in Table 1. In some cases, as "Kjkmi" running 6 threads, as well as in other experiments we have performed with larger problem sizes, we observed that the OpenMP versions were marginally better than the pure MPI version.

MPI		Threads							
Tasks	Version	0	2	3	4	5	6	7	8
12	kjmi					56.61	55.33	52.96	40.55
	Kjkmi					56.62	57.00	56.70	57.14
24	kjmi			49.87	40.39				
	Kjkmi			65.69	45.61				
48	kjmi		39.41						
	Kjkmi		45.69						
84	diag	41.68							
96	diag	40.93							

Table 3. Elapsed time in seconds, for the large problem size, running on 12 IBM SP Nighthawk I Nodes.

Table 3 presents the best elapsed time over two runs using the larger problem size (300×300×100) on 12 IBM SP Nighthawk I Nodes at Lawrence Livermore National Laboratory. Notice that this table shows only a few combinations of MPI tasks and OpenMP threads that were selected from the large space of possible configurations. Also, for each MPI task, only one decomposition was considered. Therefore, based on the results from the small problem size, where we observed that the MPI performance is heavily dependent on the decomposition, one should be aware that the MPI decomposition chosen for these experiments, was based on the observations from the small problem size, but the times might not necessarily represent the best MPI performance for this problem size.

We observe that when running the large problem size with all 96 processors, we were able to confirm the analysis derived from the small problem size for the mixed MPI/OpenMP version of "kjmi". This version performed slightly better than the pure MPI version with all three combinations of tasks and threads (namely, 48/2, 24/4, and 12/8). On the other hand, the performance of the "Kjkmi" version did not perform as well as expected, and in the only situation where its performance was comparable to the "kjmi" version, it did dot scale well with more than 5 threads. Thus, more analysis is necessary to understand its performance behavior.

When running the mixed mode approach, we observed some conflicts in the scheduling of the two parallelization strategies (MPI and OpenMP). This prob-

lem can be observed in Figure 10, which shows for each thread, the pattern of computation from the iterations of the parallel loop, when running "kjmi" with the small problem size, using 2 MPI tasks and 8 OpenMP threads per task. In this figure, where each dark (blue) area between two flags corresponds to one iteration of the loop, we can observe an unbalance between threads inside of each MPI task. The reason for this unbalance is due to the MPI pipelining that was set to have "k" dimension of 10 planes. Hence, the parallel loop had only 10 iterations, and when scheduling such number of iterations among 8 threads, two of then will perform two iterations, while the other six threads will perform only one iteration and then wait for the first two to finish.

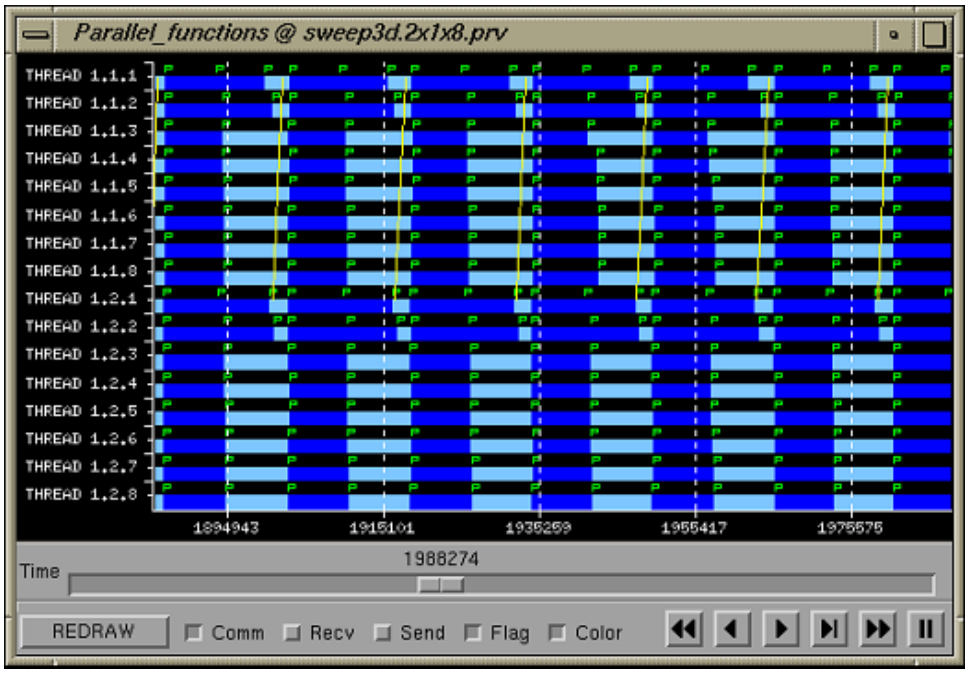

Fig. 10. Scheduling of the OpenMP loop iterations

Therefore, another important observation of this experiment was that when mixing different programming models, it is of key importance to analyze the scheduling decisions taken by the different parallelization strategies. It is fairly easy for these strategies to interfere with each other, and without an analysis tool such as OMPtrace and Paraver it may be difficult to understand the effects in performance.

5 Conclusions

In this paper we described OMPtrace, a dynamic tracing mechanism that combines traditional tracing with dynamic instrumentation and access to hardware performance counters to create a powerful tool for performance analysis and optimization of OpenMP applications. Performance data collected with OMPtrace is used as input to the Paraver visualization tool for detailed analysis of the parallel behavior of applications.

The usefulness of OMPtrace and the power of Paraver for tuning and optimizing OpenMP applications was illustrated in a case study with the US DOE ASCI Sweep3D benchmark. We analyzed the performance of a small problem size of Sweep3D, running on a single IBM SP node with 16 processors, and validated the performance observations and code optimizations, running a mixed MPI/OpenMP version of the code, with a larger problem size, on 12 IBM SP Nodes with 8 processors each.

The performance of the original OpenMP version was three times slower than the MPI version when running on 12 processors of a single IBM SP node, but when running the optimized version with the larger problem size on 96 processors, the mixed MPI/OpenMP version performed slightly better than the pure MPI version for all three combinations of MPI tasks and OpenMP threads used (48 task and 2 threads, 24 tasks and 4 threads, and 12 tasks and 8 threads).

We notice that the two dimensional MPI parallelization and the corresponding combinations of possible decompositions had an important effect on the performance of the MPI program. In OpenMP there is only one dimensional parallelization and it would be interesting to experiment with nested OpenMP parallelism.

Finally, we observe that there are still several issues, such as memory management and scheduling conflicts, that remain as challenges for optimization of the mixed MPI/OpenMP application.

References

1. B. R. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. In *Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
2. L. DeRose and T. H. Hoover Jr. and J. K. Hollingsworth The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools. In *Proceedings of 2001 International Parallel and Distributed Processing Symposium*, April 2001.
3. European Center for Parallelism of Barcelona (CEPBA). *Paraver - Parallel Program Visualization and Analysis Tool - Reference Manual*, November 2000. http://www.cepba.upc.es/paraver.
4. K. R. Koch, R. S. Baker, R. E. Alcouffe. Solution of the First-Order Form of the 3-D Discrete Ordinates Equation on a Massively Parallel Processor. In *Trans. Amer. Nuc. Soc.* 65(198), 1992.
5. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. In *IEEE Computer*, 28(11):37–46, November 1995.